Gitbook

从不会到会 Tornado TCP

Ethon

HuangSong 著

目錄

Introduction	1.1
What' the tornado?	1.2
ioloop	1.3
ioloop的基本函数	1.3.1
ioloop的回调函数	1.3.2
ioloop相关函数说明	1.3.3
what' the iostream?	1.4
BaseIOStream的相关接口	1.4.1
baseIOStream的相关函数以及IOStream类	1.4.2
Tcp Client和Tcp Server	1.4.3
一个简单的TCPServer	1.4.4
what's the RPC?	1.5
RPC on my server	1.5.1
RPC on my client	1.5.2
解读Tornado	1.6
epoll	1.6.1
epoll的CPU和内存消耗	1.6.2
tornado在TCP层里的工作	1.6.3
tornado TCPServer的设计解读	1.6.4
ioloop分析	1.6.5
让我们实战吧	1.7
正确关闭服务器的姿势	1.7.1
自动收集rpc函数	1.7.2
和数据库的那些事	1.7.3
跟多线程搞一些事情	1.7.4
tornado和celery很配呦	1.7.5
监控你的api	1.7.6

Tornado Tcp Program

本书主要通过讲解tornado相关api及技巧，来进行tcp编程以及rpc相关知识。

这里是各种服务器性能的一个简单对比，看过之后，可以发现tornado的性能还是很不错的，基本可以达到go的水平。特别是在pypy的环境下。

本书写的比较浅略，如果有兴趣，欢迎跟我 一起研究相关内容。

如果对你有帮助，请点右上角subscribe或者star.

what's the tornado?

Tornado 和现在的主流 Web 服务器框架（包括大多数 Python 的框架）有着明显的区别：它是非阻塞式服务器，而且速度相当快。得利于其 非阻塞的方式和对 epoll 的运用，Tornado每秒可以处理数以千计的连接，因此 Tornado 是实时 Web 服务的一个理想框架。

但其实tornado也非常适合写tcp服务器,在twisted和tornado的性能对比中可以发现，tornado的性能远大于twisted，尤其是在pypy的环境下。因此，本书主要来讲解如何使用tornado来进行tcp的开发。

本书的读者为具有一定python基础，使用过或想去使用tornado的读者。

本书实验环境为tornado4.3/4.4， python版本为py2.7/pypy4.0.

本书主要参考tornado官方教程以及网上的资料，如果有涉及到版权问题，请联系我。

2.ioloop

说到tornado，那就不得不说他的ioloop，这是这个框架的灵魂所在。通过一段简单的代码，来开启tornado tcp编程的大门。

```
import errno
import functools
import tornado.ioloop
import socket

def connection_ready(sock, fd, events):
    while True:
        try:
            connection, address = sock.accept()
        except socket.error as e:
            if e.args[0] not in (errno.EWOULDBLOCK, errno.EAGAIN):
                raise
            return
        connection.setblocking(0)
        handle_connection(connection, address)

if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.setblocking(0)
    sock.bind(("", port))
    sock.listen(128)

    io_loop = tornado.ioloop.IOLoop.current()
    callback = functools.partial(connection_ready, sock)
    io_loop.add_handler(sock.fileno(), callback, io_loop.READ)
    io_loop.start()
```

这是官方文档给出的一段简单的tcpserver的代码，那么我们就从它来分析一下。 首先是socket的部分，创建了一个socket，然后下面是ioloop的部分。首先我们创建一个ioloop实例，然后创建了一个回调函数，（partial这个函数不用太关心，就是一种绑定参数的函数写法。）然后给ioloop加上一个handler，用于监听socket，最后开启ioloop。当ioloop开启以后，就会去执行回调函数connecction_ready。

以上就是创建一个simpleTCPServer的步骤，接下来我们主要围绕ioloop来说一下与它相关的函数方法。

2.1 ioloop的基本函数

1.IOLoop.current(instance=True)

如果当前的ioloop已经运行了，那么这个函数就是用来获得当前线程里的IOLoop对象。

2.IOLoop.start()

这个函数就很简单了，就是开启我们的ioloop，它会一直运行下去，直到有人调用了stop()。

3.IOLoop.stop()

这个就是上面说的用来停止ioloop循环的。

4.IOLoop.run_sync(func, timeout=None)

这个函数是在ioloop开启时去执行func这个函数，然后关闭ioloop，func执行完它会自动执行stop()，这个不用担心。

```
@gen.coroutine
def main():
    # do sth...

if __name__ == '__main__':
    IOLoop.current().run_sync(main)
```

5.add_handler(fd, handler, events)

注册一个handler，从fd那里接受事件。 fd呢就是一个描述符，events就是要监听的事件。events有这样几种类型，IOLoop.READ, IOLoop.WRITE, 还有IOLoop.ERROR. 很好理解，读写事件，还有错误异常。

当我们选定的类型事件发生的时候，那么就会执行handler(fd, events)。

6.update_handler(fd, events)

用来更新上面我们注册的handler的。

7.remove_handler(fd)

停止监听fd上面的所有事件。

以上就是在ioloop开启的时候，涉及到的主要函数及其作用的介绍。

2.2 ioloop的回调函数

1.IOLoop.add_callback(callback, *args,* *kwargs)

这个函数是最简单的，在ioloop开启后执行的回调函数callback，*args*和*kwargs都是这个回调函数的参数。一般我们的server都是单进程单线程的，即使是多线程，那么这个函数也是安全的。

2.IOLoop.add_callback_from_signal(callback, *args,* *kwargs)

这个函数和上面的很类似，只不过他是在without any stack_context的时候去执行，关于stack_context，查阅这里。

3.IOLoop.add_future(future, callback)

这个函数呢也是添加一个callback函数，当给定的这个future执行完的时候，callback会去执行，这个函数有唯一的一个参数就是这个future对象。关于future呢，后面会详细去讲。

4.IOLoop.add_timeout(deadline, callback, *args,* *kwargs)

执行callback函数在deadline的时候，这个deadline可以是time.time，也可以是datetime.timedelta。还有，这个函数线程不安全。

5.IOLoop.call_at(when, callback, *args,* *kwargs)

这个函数我们用的就很多了，在ioloop启动后，会在when这个时间点去执行callback函数。类似一个定时器的功能。

6.IOLoop.call_later(delay, callback, *args,* *kwargs)

这个函数和上面的差不多，这是在ioloop启动后的delay秒后，去执行callback。上面的是一个时间点，而这个函数是多少秒。 `ioloop.IOLoop.current().call_later(3, func)` 这就是在ioloop启动的3s以后来执行func函数。

7.IOLoop.remove_timeout(timeout)

这个函数就是用来移除上面通过add_timeout注册的callback函数。

8.IOLoop.spawn_callback(callback, *args,* *kwargs)

这个函数也是去执行一个回调函数，但是和上面说过的其他callback不同，它和回调者的栈上下文没有关联，因此呢，他比较时候去做一些独立的功能回调。

9.IOLoop.time()

它返回的时候ioloop开启后的时间，返回的值跟time.time差不多。

2.3 ioloop相关函数说明

1.add_handler, update_handler, remove_handler三个和handelr有关的函数，通常是在写connection的时候用到的，刚开始学习的时候，不用太在意它的用法，在后面会详细说明。

2.add_callback是比较常用的函数之一，不管在tornado还是其他的异步框架中，回调都是非常常见的。在服务器启动以后，会有一些相关的操作需要在ioloop启动后才能有效的去执行，那么这个时候，添加一个callback就是非常必要的了。以游戏服务器为例子(可能不贴切)，在游戏服务器启动后，会把全服的排行榜数据load到内存中，那么这个时候就可以使用add_callback了。

```
io_loop = ioloop.IOLoop.instance()
io_loop.add_callback(load_rank_list)
io_loop.start()
```

其中load_rank_list就是去将排行榜信息load到内存中的相关操作。

3.call_later和call_at也是我们在开发过程中常用的函数之一。它们给我们在做相关定时的操作的时候带来了便利。举个简单的例子，在晚上9点需要向玩家发放一些奖励道具，那么我就可以用到call_at

`call_at(time, func, *args, **kw)`

time就是晚上9点的时间戳，func就是发放奖励的函数，后面是相关参数。call_later同理，在第一次发完以后，我们就可以使用call_later()来进行一个循环的操作， `call_later(86400, func)` 这样每过86400s(一天)，就可以再次发放奖励。

4.ioloop的完全开启，也就相当于我们整个服务器开启，如果想要获得服务器从开启到现在的过了多久，那么IOLoop.time()就可以帮助我们得到它。

3.what's the iostream

我们了解了关于ioloop的一些相关函数，我想现在对tornado的一些功能也有了大概的了解。这一章我们简单的了解一下tornado里的iostream。我们写一个服务器，所有的io都要通过一个connection来传输，那么iostream里就包含了我们对数据的处理，以及对连接的一些操作。

本章主要对BaseIOStream和IOStream这两个类来进行讲解。官方文档见这里。

3.1BaseIOStream的相关接口

简单的来说，这个类从socket中读或写数据。

以下是它的几个基本属性：

```
io_loop – 当前的ioloop实例。
max_buffer_size – 最大的可接受数据大小，默认是100M。
read_chunk_size – 读取的数据大小，默认64k。
max_write_buffer_size – 最大的写buffer大小。
```

这些都可以在继承的时候修改的。

介绍一下主要的几个接口

1.BaseIOStream.write(data, callback=None)

异步的写数据，如果有callback，那么在所有数据成功写入以后执行callback。

2.BaseIOStream.read_bytes(num_bytes, callback=None, streaming_callback=None, partial=False)

异步的读取数据，读到的数据大小取决于num_bytes。同理，如果有callback，在数据完全读取后，执行callback。读取到的数据data作为callback的参数，如果不是的话，那么callback需要返回一个future对象。而streaming_callback,是当读取到的数据全都是有效的情况下，才会去执行。

3.BaseIOStream.read_until(delimiter, callback=None, max_bytes=None)

同上，只不过是当读delimiter(分隔符)的时候停止。callback和read_bytes里的一样。

4.BaseIOStream.read_until_regex(regex, callback=None, max_bytes=None)

通过正则来读取数据，regex就是给定的正则表达式。

5.BaseIOStream.read_until_close(callback=None, streaming_callback=None)

异步读数据，知道socket关闭。

6.BaseIOStream.close(exc_info=False)

关闭当前的stream。

7.BaseIOStream.set_close_callback(callback)

当stream关闭时，执行回调函数。

8.BaseIOStream.closed()

如果为真，那么说明stream已经关闭了。

9.BaseIOStream.reading()

如果为真，那么说明当前正在从stream中读数据

10.BaseIOStream.writing()

有读就有写。

以上呢就是baseIOStream中，主要的接口。在我们自己去写connection的时候，都会用的到，这里先简单介绍一下，在后面写connection的时候，我们会着重去讲解。

3.2 baseIOStream的相关函数以及IOStream类

1.BaseIOStream.fileno()

这个很简单，就是我们stream当前的文件描述符。

2.BaseIOStream.close_fd()

关闭这个fd。

3.BaseIOStream.write_to_fd(data)

尝试向这个fd去写data， 期间可能会出现未成功写入，因此函数的返回值是成功写入数据的大小。

4.BaseIOStream.read_from_fd(data)

有写也有读。

5.BaseIOStream.get_fd_error()

获取fd中所有error信息。

关于baseIOStream的信息差不都就这些了，在我们实际使用的过程中，我们不会去直接使用这个类的，我们基本都会去使用它的子类，IOStream。

以下通过官方文档的例子，来简单介绍一下IOStream。

```
import tornado.ioloop
import tornado.iostream
import socket

def send_request():
    #向目标写数据
    stream.write(b"GET / HTTP/1.0\r\nHost: friendfeed.com\r\n\r\n")
    #读数据，执行回调on_headers
    stream.read_until(b"\r\n\r\n", on_headers)

def on_headers(data):
    headers = {}
    for line in data.split(b"\r\n"):
       parts = line.split(b":")
       if len(parts) == 2:
           headers[parts[0].strip()] = parts[1].strip()
    #读数据，最后关闭stream，ioloop
    stream.read_bytes(int(headers[b"Content-Length"]), on_body)

def on_body(data):
    print(data)
    stream.close()
    tornado.ioloop.IOLoop.current().stop()

if __name__ == '__main__':
    #创建一个socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    #创建一个stream
    stream = tornado.iostream.IOStream(s)
    #连接目标，执行回调函数send_request
    stream.connect(("friendfeed.com", 80), send_request)
    开启ioloop
    tornado.ioloop.IOLoop.current().start()
```

3.3 Tcp Client和Tcp Server

tornado.tcpclient.TCPClient(resolver=None, io_loop=None)

这个呢就是tornado自有的一个tcpclient，使用它的时候，可以直接继承它.

它有一个方法 `connect(host, port, af=<AddressFamily.AF_UNSPEC: 0>, ssl_options=None, max_buffer_size=None)`

它会返回IOStream的实例，如果ssl_options为眞，那么会返回SSLIOStream。

通过给的host和port来连接服务器，然后通过返回的stream，就可以进行读写等操作了。

tornado.tcpserver.TCPServer(io_loop=None, ssl_options=None, max_buffer_size=None, read_chunk_size=None)

这个就是用来创建TCPserver的。它是非阻塞，单线程的。

来看一下关于它的几个函数

1.listen(port, address="")

```
server = TCPServer()
server.listen(8888)
IOLoop.current().start()
```

这就可以创建一个简单的tcpserver，端口号8888

2.bind(port, address=None, family=, backlog=128)

开启多进程的一个方法

```
server = TCPServer()
server.bind(8888)
server.start(0)  # Forks multiple sub-processes
IOLoop.current().start()
```

3.add_sockets(sockets)

也可以开启多进程。

```
sockets = bind_sockets(8888)
tornado.process.fork_processes(0)
server = TCPServer()
server.add_sockets(sockets)
IOLoop.current().start()
```

4.handle_stream(stream, address)

这是我们用来接收stream的方法。你可以通过继承TCPServer，来覆盖这个方法。

几个方法非常简单，都是最基本的，在写server的时候都需要用到的。下一节我们来写一个简单的tcpserver。

3.4 一个简单的TCPServer

以下是一个简单的TCPServer的代码(鸣谢yoki123)。

```
class TcpServer(object):
    def __init__(self, address, build_class, **build_kwargs):
        self._address = address
        self._build_class = build_class
        self._build_kwargs = build_kwargs

    def _accept_handler(self, sock, fd, events):
        while True:
            try:
                获得conn
                connection, address = sock.accept()
            except socket.error, e:
                return

            #通过conn解析
            self._handle_connect(connection)

    def _handle_connect(self, sock):
        #这里的conn主要是我们来解析数据的protocol
        conn = self._build_class(sock, **self._build_kwargs)
        self.on_connect(conn)

        close_callback = functools.partial(self.on_close, conn)
        #设置一个conn关闭时执行的回调函数
        conn.set_close_callback(close_callback)

    def startFactory(self):
        pass

    def start(self, backlog=0):
        #创建socket
        socks = build_listener(self._address, backlog=backlog)

        io_loop = ioloop.IOLoop.instance()
        for sock in socks:
            #接受数据的handler
            callback = functools.partial(self._accept_handler, sock)
            #为ioloop添加handler，callback
            io_loop.add_handler(sock.fileno(), callback, WRITE_EVENT | READ_EVENT | ER
ROR_EVENT)
        #在ioloop开启后，添加一个回调函数
        ioloop.IOLoop.current().add_callback(self.startFactory)

    #接受buff的函数，继承tcpserver的时候可以重写。
    def handle_stream(self, conn, buff):
```

```
            logger.debug('handle_stream')

    def stopFactory(self):
        pass

    def on_close(self, conn):
        logger.debug('on_close')

    def on_connect(self, conn):

        logger.debug('on_connect: %s' % repr(conn.getaddress()))

        handle_receive = functools.partial(self.handle_stream, conn)
        conn.read_util_close(handle_receive)
```

我们来看一下流程，

1.start().我们开启**tcpserver**，首先创建一个**socket**，**,** 然后我们为**ioloop**添加**handler**。

2.服务器开启后，执行**_accept_handler,**这个函数里是一个循环，从**socket**中获取**conn**

3.然后我们解析这个**conn**， **build_class**使我们自己实现的一个用来解析数据，拆包解包的一个**protocol**类，这里我们会将所有的**bytes**解析为我们能看懂的字符串数字等等。

4.然后我们调用**on_connect**函数，我们将这个**conn**传入**handler_stream**中， 前面讲过，这个函数是用来接受的**stream**的。然后我们从**stream**中读取数据，知道**conn**关闭。

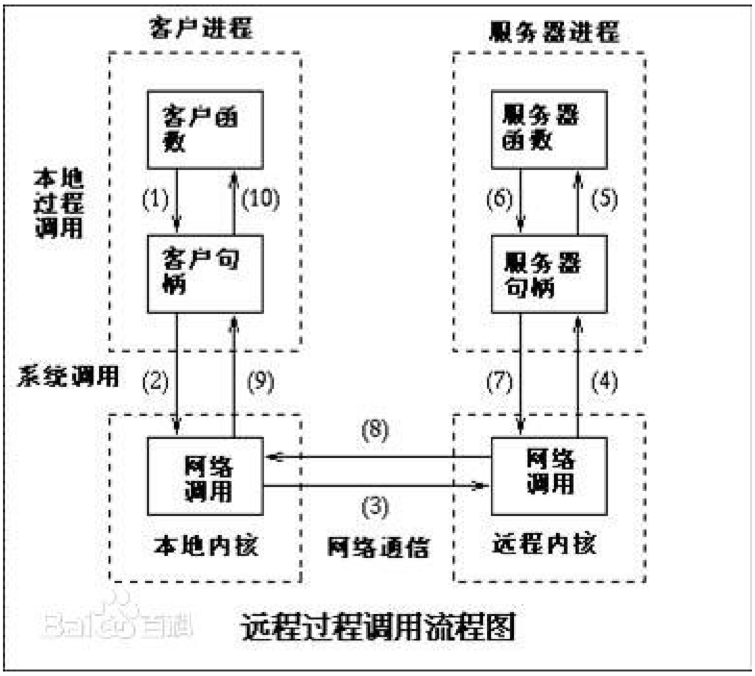
以上就是我们服务器开启后执行的大概流程。

what's the RPC?

RPC（Remote Procedure CallProtocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

简单来说**rpc**就是**client**在不知道任何底层实现的情况下，可以直接调用**server**的函数方法；而**server**也可以直接调用**client**的函数。

百度给出的流程图：



简单了解了什么是**rpc**，下面我就来在我们的**server**和**client**来实现**rpc**的功能。

RPC on my server

下面是之前我们给出的**tcpserver**中的一个函数，用来处理连接

```
def _handle_connect(self, sock):
    conn = self._build_class(sock, **self._build_kwargs)
    self.on_connect(conn)

    close_callback = functools.partial(self.on_close, conn)
    conn.set_close_callback(close_callback)
```

build_class之前我们说过，这是我们用来处理数据的**protocol**，那么**rpc**的逻辑流程应该都写在这里。现在假设**client**来调用一个函数**sum(x, y)**， 那么在我们**server**中就要有这样一个函数。

```
def sum(x, y):
    return x+y
```

但是客户端来调用，我们如何知道server有这样一个函数呢？这就需要我们提前去处理一下，将希望可以被client调用的函数存起来。

```
def route(**options):
    def decorator(handler):
        msgid = options.pop('msgid', handler.__name__)
        elif not msgid in HANDLERS:
            HANDLERS[msgid] = handler
        else:
            raise Exception('[ ERROR ]Handler "%s" exists already!!' % msgid)
        return handler
    return decorator
```

比如我们可以写这样一个函数，作为装饰器，来将被装饰的函数存起来(HANDLERS), 这样在client调用sum的时候，我们就可以知道，server中是否存在sum这个函数。

```
@route()
def sum(x, y):
    return x+y
```

这样函数被装饰起来以后，我们就可以找到这个函数了。当我们知道存在sum这个函数的时候，我就可以从HANDLERS中讲sum取出来， `sum = HANDLERS.get('sum')` , 然后我们讲x，y两个参数传进去，算出结果，再将结果写回conn中，返回给client。至此，客户端调用服务器函数的大致流程就说完了。

RPC on my client

服务器调用**client**函数也是一样的道理，**client**中也要和服务器一样，将想要被调用的函数收集起来。当然这个时候，服务器还要做一件事情，就是将客户端的**tcp**连接通过某种条件储存起来，这样才知道我想去调用哪个**client**的函数。

现在有很多**client**来连接我们**server**，以游戏为例，每个玩家都有自己对应的**uid**，那么我们就可以将**conn**根据**uid**储存起来。 `client_conn = {uid:conn, ..}` 这样，我们想给哪位玩家发送消息或者做一些其他的事情，就很容易了，只要客户端写好函数功能，我们只要将函数名字，以及对应参数，通过**client_conn**找到对应玩家的**tcp**连接，然后将上述数据发送过，那么当客户端计算出结果以后，我们通过一个回调函数，就可以将结果**return**到服务器了。

现有的rpc框架很多，比如msgpack-python, 这里有很多语言的rpcSimpleServer，感兴趣的同学可以参考一下。

解读Tornado

前几章主要是针对TCP编程中，tornado中常用的函数进行简单的解读。本章主要对tornado性能进行分析，来看一看我们到底为什么要在tornado的基础上进行tcp编程。

epoll

我们知道tornado通过非阻塞的方式以及对epoll的运用，才使得性能上得到了很大的提升。那么epoll到底是什么，它在tornado中扮演着怎样的角色呢？

1.epoll解读

说到epoll，就得先说说阻塞和非阻塞，这里大家自行百度或者脑补。 我们通常处理数据流可能是这样的

```
While true：
    for i in stream:
        if i has data:
            Do something with i
```

这种方式显然很差劲，他会一直轮询，不管数据流中是否有IO事件。针对这种情况就出现了select， 它可以甄别数据流是否有IO，当无IO的时候，就阻塞在那里，直到下一次IO发生，在进行操作。

```
While true：
    select（stream）
    for i in stream：
        if i has data：
            Do something
```

虽然我们不用白白的轮询，也知道了是否发生IO，，但却并不知道是那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。 因此epoll就诞生了，epoll全称就是event poll，和咱们平常用的轮询不同，基于事件的epoll会把哪个数据流发生了怎样的事件告诉我们。

```
while true
    active_stream[] = epoll_wait(epollfd)
    for i in active_stream[]：
        read or write till unavailable
```

2.epoll的性能优势

前面说了epoll的特点，那么这些特点能带来什么好处呢？Epoll在绝大多数情况下性能都远超 select 或者 poll，但是除了速度之外，三者之间的 CPU 开销，内存消耗情况又怎么样呢？

以下来自**stackoverflow**网友的问答翻译

问**:**

我读过所有的关于**tornado**的书告诉我在**epoll**是可以替代**select**和**poll**的，特别是对**twisted(twsited**是**python**的另一个十分有名的网络库**)**。查阅**epoll**以及其他的**(select**等**)**相关资料表明，**epoll**速度很快并且拓展性很强，这是否表明在**CPU**和内存消耗上，**epoll**仍然很强？

答：

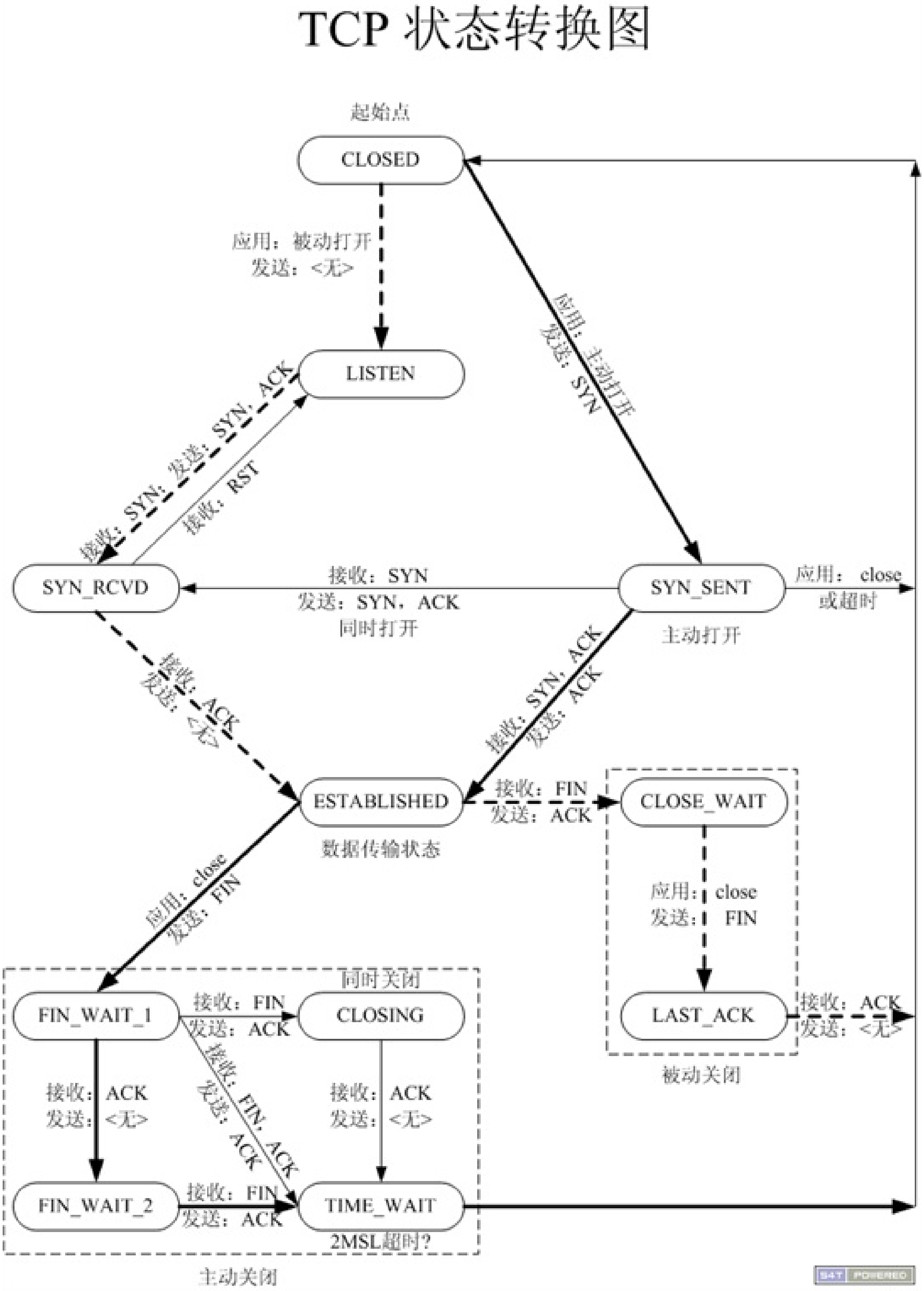
在**socket**数量很少的时候，**select**在内存消耗以及运行效率上都超过**epoll**，当然，这个差距是非常小的，以至于在绝大多数环境下都可以忽略。但是，无论是选择**select**还是**epoll**，在不同场景下面临的**api**复杂度是不一样的。如果你选择了一个单一的**fd**，**100**，那么它的性能会高于两倍以上的**fd**，**50.**

Epoll的成本接近于**fd(**文件描述符**)**，如果你监控**200**个**fd**，其中只有**100**个有**IO**事件，那么你只需要支付这**100**个**fd**的消耗，而不用在乎另外的**100**个，这就是**epoll**提供的优势。相反，当你使用**select**的时候，如果**1000**个**fd**都处于空闲，你仍然要监控这**1000**个**fd(**简单的来讲，就是**epoll**只为有**io**的**fd**支付。你干活，我付钱给你；你不干活，那么我就不给你钱**)**。那么这就意味着更少的**CPU**使用率。

关于内存使用率，简单来说，使用**select**的话，你要花**384**字节监视一个文件描述符，但它的价值是**1024**，而用**epoll**你只花**20**个字节。不过，所有这些数字都很小，所以没有太大的区别。

tornado 在TCP里的工作

首先是关于 TCP 协议。这是一个面向连接的可靠交付的协议。由于是面向连接，所以在服务器端需要分配内存来记忆客户端连接，同样客户端也需要记录服务器。为了安全所以有了三次握手机制，这里给出一张图-- 状态转换图(UNIX网络编程)



对于 TCP 编程的总结就是：创建一个监听 socket，然后把它绑定到端口和地址上并开始监听，然后不停 accept。这也是 tornado 的 TCPServer 要做的工作。

TCPServer 类的定义在 tcpserver.py。它有两种用法：bind+start 或者 listen。

简言之，基于事件驱动的服务器（tornado）要干的事就是：创建 socket，绑定到端口并 listen，然后注册事件和对应的回调，在回调里accept 新请求。

创建监听 socket 后为了异步，设置 socket 为非阻塞（这样由它 accept 派生的socket 也是非阻塞的），然后绑定并监听之。add_sockets 方法接收 socket 列表，对于列表中的 socket，用 fd 作键记录下来，并调用add_accept_handler 方法。它也是在 netutil 里定义的。

add_accept_handler 方法的流程：首先是确保ioloop对象。然后调用 add_handler 向 loloop 对象注册在fd上的read事件和回调函数accept_handler。该回调函数是现成定义的，属于IOLoop层次的回调，每当事件发生时就会调用。回调内容也就是accept得到新socket和客户端地址，然后调用callback向上层传递事件。从上面的分析可知，当read事件发生时，accept_handler被调用，进而callback=_handle_connection被调用。

TcpServer类的解读

在**TCPServer**类的注释中，首先强调了它是一个**non-blocking, single-threaded TCP Server.**

那么如何理解non-blocking呢？ non-blocking，就是说，这个服务器没有使用阻塞式API。 通常来说，我们socket的读写都是阻塞式的,不管有没有数据，服务器都派API去读，读不到，API就不会回来交差。而非阻塞区别在于没有数据可读时，它不会在那死等，它直接就返回了。 而single-thread，说的是服务器是单线程模式，一个线程可以监视成千上万的连接，因此不需要多线程。在ubuntu上用的是epoll，bsd用的是kqueue。

在使用方面，tcpserver这个类一般不直接使用，而是派生出子类，然后让子类实例化。可以看到作者是强制去继承tcpserver，handle_stream并没有实现，如果你不去覆盖，就会报错。

```
def handle_stream(self, stream, address):

    """Override to handle a new `.IOStream` from an incoming connection."""

    raise NotImplementedError()
```

ioloop分析

咱们先来简单说一下**ioloop**的源码

```
while True:
    poll_timeout = 3600.0

    # Prevent IO event starvation by delaying new callbacks
    # to the next iteration of the event loop.
    with self._callback_lock:
        #上次循环的回调列表
        callbacks = self._callbacks
        self._callbacks = []
    for callback in callbacks:
        #执行遗留回调
        self._run_callback(callback)

    if self._timeouts:
        now = self.time()
        while self._timeouts:
            #超时回调
            if self._timeouts[0].callback is None:
                # 最小堆维护超时事件
                heapq.heappop(self._timeouts)
            elif self._timeouts[0].deadline <= now:
                timeout = heapq.heappop(self._timeouts)
                self._run_callback(timeout.callback)
            else:
                seconds = self._timeouts[0].deadline - now
                poll_timeout = min(seconds, poll_timeout)
                break

    if self._callbacks:
        # If any callbacks or timeouts called add_callback,
        # we don't want to wait in poll() before we run them.
        poll_timeout = 0.0

    if not self._running:
        break

    if self._blocking_signal_threshold is not None:
        # clear alarm so it doesn't fire while poll is waiting for
        # events.
        signal.setitimer(signal.ITIMER_REAL, 0, 0)

    try:
        #这里的poll就是epoll，当有事件发生，就会返回，详情参照tornado里e#poll的代码
        event_pairs = self._impl.poll(poll_timeout)
    except Exception as e:
```

```
            if (getattr(e, 'errno', None) == errno.EINTR or
                (isinstance(getattr(e, 'args', None), tuple) and
                 len(e.args) == 2 and e.args[0] == errno.EINTR)):
                continue
            else:
                raise

        if self._blocking_signal_threshold is not None:
            signal.setitimer(signal.ITIMER_REAL,
                             self._blocking_signal_threshold, 0)

        #如果有事件发生，添加事件，
        self._events.update(event_pairs)
        while self._events:
            fd, events = self._events.popitem()
            try:
                #根据fd找到对应的回调函数，
                self._handlers[fd](fd, events)
            except (OSError, IOError) as e:
                if e.args[0] == errno.EPIPE:
                    # Happens when the client closes the connection
                    pass
                else:
                    app_log.error("Exception in I/O handler for fd %s",
                                  fd, exc_info=True)
            except Exception:
                app_log.error("Exception in I/O handler for fd %s",
                              fd, exc_info=True)
```

简单来说一下流程，首先执行上次循环的回调列表，然后调用**epoll**等待事件的发生，根据**fd**取出对应的回调函数，然后执行。**IOLoop**基本是个事件循环，因此它总是被其它模块所调用。

咱们来看一下**ioloop**的**start**方法，**start** 方法中主要分三个部分：一个部分是对超时的相关处理；一部分是 **epoll** 事件通知阻塞、接收；一部分是对 **epoll** 返回**I/O**事件的处理。

1.超时的处理，是用一个最小堆来维护每个回调函数的超时时间，如果超时，取出对应的回调，如果没有则重新设置poll_timeout的值

2.通过 self._impl.poll(poll_timeout) 进行事件阻塞，当有事件通知或超时时 poll 返回特定的event_pairs，这个上面也说过了。

本章主要介绍一些开发中的小例子，小问题，小技巧。

1.正确关闭服务器的姿势

有时候，服务器的进程因为某种原因被关闭或者自己手动关闭，无法保证内存里的数据正确入库或者仍有**cb**函数没有执行，这个时候，我们就要确保一切都正确的被执行完毕后，再关闭服务器。

```
def sig_handler(sig, frame):
    logger.warning('Caught signal: %s', sig)
    ioloop.IOLoop.instance().add_callback(shutdown)

def shutdown():
    io_loop = ioloop.IOLoop.instance()
    server.stopFactory() ##自己去做一些处理，保证入库等。

    deadline = time.time() + 5

    def stop_loop():
        now = time.time()
        if now < deadline and io_loop._callbacks:
            io_loop.add_timeout(now + 1, stop_loop)
        else:
            io_loop.stop() # 处理完现有的 callback后，结束ioloop循环
    stop_loop()

def start():
    log_initialize()
    global server
    server = RPCServer(('localhost', 5700))
    server.start()

    signal.signal(signal.SIGTERM, sig_handler)
    signal.signal(signal.SIGINT, sig_handler)
```

2.自动收集rpc函数

服务器可被客户端调用的函数，不会有多少就去写多少到字典中，所以需要自动去收集可被调用函数,这样就方便许多。

```
collect.py

#1.第一种方法也比较傻，适合rpc文件较少的情况
import rpc1

DICT = {}

for name in dir(rpc1):
    if name.startswith("__") or name.endswith("__"):
        continue
    DICT[name] = getattr(rpc1, name)

rpc1.py

#被调用函数
def func():
    return 1
```

```
#第二种相对智能一些
将所有可被调用函数的文件写入固定文件夹中，比如取名为handler的文件夹。直接将模块import或者也可以写入一个字典中
#name为当前文件与handler的相对路径
_imported = []
for f in os.listdir(name + "/handler"):
    if f.find('.pyc') > 0:
        _subfix = '.pyc'
    elif f.find('.pyo') > 0:
        _subfix = '.pyo'
    elif f.find('.py') > 0:
        _subfix = '.py'
    else:
        continue

    fname, _ = f.rsplit(_subfix, 1)
    if fname and fname not in _imported:
        _handlers_name = '%s.%s' % (module, fname)
        __import__(_handlers_name)
        _imported.append(fname)
```

3.和数据库的那些事

在开发中,数据库是必不可少的, 因此这节主要来说一下常用的两种类型数据库,**mysql**和**redis**的简单使用

```
#1.mysql算是最常用的数据库之一了,不要钱,功能齐全,性能优良。这里主要使用tornado的一款api,
#tornado_mysql。

from tornado_mysql      import pools
from tornado            import gen
from tornado            import ioloop
from tornado.concurrent import Future
from pool               import threadpool

import functools

SYNC, ROW, DATASET = range(3)

__pool = None

ioloop = ioloop.IOLoop.current()

def init(**conf):
    global __pool

    if not __pool:
        __pool = pools.Pool(conf, max_idle_connections=5, max_recycle_sec=3)

@gen.coroutine
def execute(sql, value=None, operator=SYNC):
    assert __pool is not None

    result = None
    if value is None:
        cur = yield __pool.execute(sql)
    else:
        yield __pool.execute(sql, value)
    if operator == ROW:
        result = cur.fetchone()
    elif operator == DATASET:
        result = cur.fetchall()
    raise gen.Return(result)

fetchone = functools.partial(execute, operator=ROW)
fetchall = functools.partial(execute, operator=DATASET)
#对几种简单的数据库操作进行了简单的封装,便于开发中的使用。
```

```
#2.redis算是比较常用的数据库之一，一般使用来做cache,也有做消息队列的。这里用的是
#tornadoredis这款api。
```

```
import tornadoredis
from tornado import gen
from tornado.concurrent import Future

pool = None

def init():
    global pool

    if not pool:
        CONNECTION_POOL = tornadoredis.ConnectionPool(max_connections=500, wait_for_av
ailable=True)
        pool = tornadoredis.Client(connection_pool=CONNECTION_POOL, selected_db=12)

@gen.coroutine
def close():
    if pool:
        yield pool.disconnect()

@gen.coroutine
def batch(commands):
    assert pool is not None
    result = None
    try:
        pipe = pool.pipeline()

        for _cmd in commands:
            _op   = _cmd[0]
            _args = _cmd[1]

            if len(_cmd) > 2:
                _kwargs = _cmd[2]
            else:
                _kwargs = {}

            getattr(pipe, _op)(*args, **kwargs)

        result = yield gen.Task(pipe.execute)
    except Exception as e:
        print e

    raise Return(result)

def execute(cmd, *args, **kwargs):
    assert pool is not None
    f = Future()
    def onResult(result):
        f.set_result(result)

    result = None
    _func  = getattr(pool, cmd)
    _func(callback=onResult, *args, **kwargs)
```

```
    return f

#对redis的操作进行了统一封装,直接使用execute就可以, 将命令作为参数传递。多个命令的执行可以使用batch来执行, 使用pipe提高执行效率。
```

跟多线程搞一些事情

至今我仍然坚信，单线程仍然是最有效率的方式，比如nginx就是个很好的例子，在python中尤其。但是tornado依然提供了多线程方式来给开发者。这里给出一个例子，并做出简单的解释。

```
EXECUTOR = ThreadPoolExecutor(max_workers=4)＃最大线程数

def unblock(f):
    @tornado.web.asynchronous
    @wraps(f)
    def wrapper(*args, **kwargs):
        self = args[0]
        def callback(future):
            self.write(future.result())#将f结果写回给client
            self.finish()

        EXECUTOR.submit(
                partial(f, *args, **kwargs)#启用多线程
                ).add_done_callback(#因为是异步执行，所以返回值是future
                        lambda future: tornado.ioloop.IOLoop.instance().add_callback(
                            partial(callback, future)))#最后写回client的步骤要在主线程中完
成，否则会出错，因此需要通过回调来将f返回的future返回到主线程中。
    return wrapper

class MainHandler(tornado.web.RequestHandler):
    @unblock
    def get(self):
        sleep(3)
        #self.write("ff")
        return "ff"
```

线程库用的是futures里提供的，比较简单，没有安装的童鞋，可以用pip install futures 来进行安装. 上面的例子用的是最简单的http请求，tcp中也是同样一个道理，只需要讲self.write和finish替换成自己写socket的函数就可以了。

另外由于GIL的限制，多线程并不能达到利用多核的目的，因此还是要使用传统的多进程来实现，或者也可以使用比较流行的协程提高效率。最理想的情况就是netty那样的模型，但是因为python线程的问题，因此可以将线程替换为stackless中的微进程，每一条连接分配一个stackless，这样也可以达到一个目的。现成的有人将stackless与twisted在一起使用，具体效果没有测试过。

tornado和celery很配呦

Celery 是一个简单、灵活且可靠的，处理大量消息的分布式系统，并且提供维护这样一个系统的必需工具。它是一个专注于实时处理的任务队列，同时也支持任务调度。在我们日常的开发中，或多或少都会用到，一些比较耗时的异步任务，一些定时的任务都可以用celery去做。

在tornado中如果想使用celery，首先要安装celery的python api。使用pip install celery就可以了，非常方便。

```
from celery import Celery, task

c = Celery()

这是最基本的用法。

我们想要定义一个任务，我们就可以写一个很普通的方法，比如

@task
def mytask(a):
    print 1111

只要加上@task这样一个装饰器，它就会标识为一个celery的task。我们就可以调用他了。

首先我们要启动celery，下面是我的启动命令
celery -A my.utils.async_task worker -P gevent -c 2 -l info -n 'my.worker.%%h.%(ENV_US
ER)s'

相关参数官方文档中都可以查到，这里就不一一详述了。

调用的时候，我们要这样

v = mytask.apply_async(111, countdown=1)
他的返回值是他的任务id，通过任务id，我们可以取消任务,countdown代表多少秒以后执行

revoke函数就是取消任务的，revoke(task_id)
```

监控你的api

influxdb grafana